Assignment 07: Static Variables, Default Parameters and Function Overloading

COSC 1437: Programming Fundamentals II

Objectives

- Learn difference between static and automatic variables.
- Use static variables to implement a yield function that remembers and yields next value in a generated sequence
- Use default parameters for a function.
- Practice function overloading and learn how overloading of a function works by using function signature to determine which function is called.

Description

In this assignment we will continue to learn about creating our own user defined functions. We have a couple of concepts that you will be demonstrating and using with C++ functions in this assignment.

We start by looking at statically declared variables. In C/C++ (and many programming languages) we can divide the way variables are created in memory into two categories. Some variables are automatically created/allocated when a function is entered, and they are automatically destroyed or deallocated when the function ends. Our course text refers to these as **automatic** variables. These work by using a function call stack. Every time a function is called, space is allocated on the function call stack in order to hold the parameter values and all of the variables declared locally inside of that function. Automatic variables created on the function call stack are necessary to support recursive functions, which we will briefly look at in this class later.

The other way variables are created is referred to as static allocation. These variables are created on the program heap. Any global variable or constant is automatically a static variable, that is created when the program starts, and the space is allocated in the program heap data structure. But you can also declare variables inside of functions to be `static` as well. This allows for the creation of a variable that can only be used inside of the function (the name is local to the function). But it behaves like a global statically allocated variable. So if you modify the value in the function, the value will remain the same if the function is called again.

In this class you should never define or use global variables. This is usually considered bad practice, and we do not use them in any of our assignments. Though globally defined constants can be useful, and we may use them at times in this class. But static variables can be needed for some types of function behavior, which require a statically allocated bit of memory that is not deallocated when the function is done, but that can only be referenced and used inside of the function that declares the static variable.

When working with static variables in the first task, we will also use an example of defining a default parameter for a C++ function. This allows us to declare an input parameter and assign it a default value. The effect is that, if the person calling the function does not provide a value for the parameter with a default, then the default value will be passed in for that parameter.

The other major concept introduced in this assignment is function overloading. In the C++ language, we can actually have many functions with the same name. The only requirement for overloading a function name is that the function signature has to differ. In practical terms this means that we can overload a function, but the types (or number) of the input parameters will have to be different. Also the output return type will often be different as well.

Function overloading can be useful, though as we will see in this assignment we would really like a mechanism like **templates**, that we will cover later on, to define a whole class of functions that can be used on different input and output data types.

Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

File Name	Description
`src/assg07-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`src/assg07-library.cpp`	File that contains the code implementations, all your work will be done here
`include/assg07-library.hpp`	Header include file of function prototypes

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for 'Assignment 03' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Lab Task 1 / In-Lab Work : Yielding the Fibonacci Sequence

You may have run across the Fibonacci sequence at some point in your academic career. The Fibonacci sequence is useful in many contexts in the sciences and mathematics. It is an example of an exponentially growing series. It is usually defined as follows:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

where each term in the sequence is the sum of the previous two terms. The recurrence relationship that defines this sequence is usually written as:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

For this task you are going to write a `nextFibonacciNumber()` function. Usually we might define a function that takes a parameter `n` as input, and it calculates and returns the n^{th} Fibonacci number in the sequence. However in this case, we want a different kind of function. The function we write here will yield the next Fibonacci number in the sequence each time it is called. So it will need to remember where it is in generating the sequence, to generate the next number whenever it is called.

So declare a function named `nextFibonacciNumber()` and implement it to pass the tests given for task 1. The function should return an integer result each time it is called. The function will need to take a `bool` parameter as input, called `resetSequence`. But this should be a default parameter that defaults to a value of `false`. If you look

at the tests, we will pass in `true` the first time we call this function, to reset the sequence from the beginning. But usually we don't pass in any parameter, in which case it should default to `false`.

You will need to create two variables of type integer that are declared to be allocated statically. These will hold and remember the previous two values in the sequence, from which we can compute the next value in the sequence. For example, it is suggested you name these local static variables `F_1` and `F_2` respectively to remember the F_{-1} and F_{-2} previous two values of the sequence.

When the `resetSequence` parameter is `true`, you should initialize `F_2` to 0 and `F_1` to 1, to represent the base case initial two values. When the sequence is reset you should return 1. We don't generate and yield the 0 in this function, we start with the 1's for the first two calls to the function (see the tests).

Whenever the `resetSequence` is false, you need to calculate and return the next value in the sequence. You have the previous two values, so you can calculate the next value. However you also have to shift all of the values before returning. So after calculating the next value in the sequence, and before returning, shift the previous to the 2nd previous, and shift the next you just calculated to the previous remembered value of the sequence.

To complete the work for this task, perform the following steps:

1. As usual start by declaring the prototype of this function and adding to the assignment header file, then a stub implementation that returns 0 in the implementation file, and then enable the task 1 tests and ensure code compiles and runs.
2. Make sure you are passing in a `bool` parameter to this function that defaults to `false`.
3. Create two statically allocated local variables to hold the two previous remembered values of the sequence.
4. When the reset flag is false, you should initialize the static variables to 0 and 1 for the start of the sequence, and return 1.
5. When the reset flag is true you need to generate and return the next value of the sequence using the remembered previous two values:
    - Also before you return you have to shift the values to now remember the one you just generated and the one before that as the previous 2 values of the sequence.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 2: Overload `swap()` `threeSort()` Function on `char` Types

In tasks 2 and 3 our goal is to learn a bit about function overloading, as well as continue practicing declaring and reusing functions. In a previous assignment you implemented a `swap()` and `threeSort()` function. In fact an implementation has been given to you in this assignment that swaps and sorts integer parameters, like you should have implemented in your previous assignment.

What if we need to be able to swap and three sort variables of any data type? For example, what if in addition to being able to swap and sort `int` types, we need to be able to swap and sort characters, strings, or any other type?

As a first step towards that goal, we can always overload the functions for the types we need it to be able to handle. So for this task 2, add in new versions of the `swap()` and `threeSort()` functions. But this time, both of these function should take references to `char` types as the inputs to be swapped and sorted. As before, these have to be passed in by reference. But besides changing the types passed in, the implementation should still work the same. For example, you can compare two characters, e.f. `if (a > b)` and this returns true if the character in `a` comes alphabetically (in the ASCII character set) before `b`, and false if not.

Do the following steps for this task:

1. Enable the task 2 tests and create declarations and implementations of `swap()` and `threeSort()` functions.
2. You can copy and paste the existing versions to create your overloaded versions. The only difference is that the functions need to operator on `char` references instead of `int` references like before.
3. This is the first time you were not given the function documentation. You should also copy the function documentation, but modify it to correctly document the type being manipulated in this overloaded version that works on character types.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 2 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 3: Overload `swap()` `threeSort()` Function on `string` Types

If you successfully complete task 2 and get it to pass all tests, you should hopefully find this task trivial, as we are doing more of the same again.

The one wrinkle is that you should now overload the `swap()` and `threeSort()` functions to work on `string` reference parameters. We will be covering the C++ `string` type in a bit more detail later. But this type allows you to represent and compare strings. So for example if you look at the tests for task 3, the first swap test looks like this:

```
// to pass by value, we need local variables
string a;
string b;

// swap two values
a = "California";
b = "Michigan";
swap(a, b);
CHECK(a == "Michigan");
CHECK(b == "California");
```

So from this you can see that variables `a` and `b` are of type `string` here. You can compare strings using the boolean operators, so the `==` will be true here if the strings are the same, and we are testing that they get swapped successfully after calling your overloaded function.

And BTW, comparisons work as you would expect, so `a < b` would be true if the string in `a` comes alphabetically before `b` and false if not. And you can also assign strings just like basic types, so you can implement the moving around of values in the swap in the same way as before.

So the steps here are the same as for the previous task, just overload a new set of functions to work on `string` types:

1. Enable the task 2 tests and create declarations and implementations of `swap()` and `threeSort()` functions.
2. You can copy and paste the existing versions to create your overloaded versions. The only difference is that the functions need to operate on `string` references instead of `int` or `char` references like before.
3. This is the first time you were not given the function documentation. You should also copy the function documentation, but modify it to correctly document the type being manipulated in this overloaded version that works on character types.

When finished, save, compile and rerun the tests. Once you are satisfied, make and push your commit of Task 3 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 50 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 25 additional points are given by the autograder for completing tasks 2 and 3 successfully.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request

comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax